Virtual Registers

R. W. Doran, P. McA. Fenwick, Zheng Qun

Computer Science Department
University of Auckland

Abstract

This paper concerns computer architectures and computer designs in which registers are virtual, that is, where registers do not necessarily correspond exactly to a real fast storage array, in particular where there are many more registers addressable than fast storage elements implemented. There are many architectures where registers are to a greater or lesser extent virtual. One scheme, where the in-use registers are managed as a set associative cache, is explored in some depth - it is seen that such a scheme is not as unreasonable as it might first appear.

Introduction

In order to avoid confusion, let's commence by carefully defining our terms. The *functional architecture* of a computer is what the compiler/machine language user sees, the level of the instruction set or the Principles of Operation. The level immediately beneath the functional architecture, how the functional architecture is implemented in terms of logical circuits is its *design architecture*. Where no confusion will result we will use *architecture* for functional architecture and *implementation* or *design* for the lower levels.

The use of architectural terminology is often very imprecise because the boundary between the two levels of description is itself very fuzzy. The functional architecture may well reflect features of the design; indeed this can be taken to the extent of eliminating the functional architecture entirely. There are often clear performance advantages to be obtained by allowing software to be aware of design architecture. However, it is generally regarded as desirable to keep the two levels independent of each other as much as is possible. The overall reason for independence is that the one functional architecture may have many implementations at different performance levels and will bridge many different generations of technology if it is successful - the difficulties involved in requiring different software for each design is unconscionable.

Although there is a desire by architects to separate architecture from design it is in practise difficult to keep them apart yet achieve a reasonable performance/cost with any particular technology. One has to compromise; architecture, is one of the "arts of the possible". Indeed, the everyday architectures are so shot-through with concessions to design that it is obvious that the machine that we are dealing with is far from the ideal. Some of these features are so prevalent that we tend to forget that they are compromises rather than givens. For example:

- limited precision operands
- limited capacity storage
- fixed-length storage locations
- use of addresses rather than names for operands
- two levels of storage - registers and main

For most of these features there certainly have been attempts made to provide higher-level alternatives, but these have not affected the main stream of development unless they are made in a spirit of compromise. An example would be the use of names in capability machines (which are now ubiquitous given that the IBM System/370 ESA and AS400 have capability features) but only within certain situations, addresses still being used where performance is important.

Here we will explore the issue of registers which are usually regarded as programmer-managed high-speed storage. The situation with registers is interesting in that an intermediate level in memory, the cache, has been introduced into most modern designs without its presence appearing in the architecture. Thus we find the same device being introduced into adjacent levels of an architecture hierarchy whereas one of the purposes of layered specification is to eliminate such duplication. We will explore the question as to whether it is really necessary that registers appear in architectures, or whether there are other compromises available.

The Purpose of Registers

When one considers the reason for the appearance of registers in functional architectures, the first thought is that they are there because they are *fast*, which is why the question of overlap of function with cache arises. However, registers have another raison d'être, which is that, because they are few in number, they can be addressed with few bits. If they are used to identify quantities that are used frequently by a program they can allow programs to be made much more compact. For example, the AS400 from IBM in its high-level architecture has instructions that implement functions like:

A[i] := B[j] + C[k] (full indexing) or x := y + z (direct addressing)

where each quantity identified is an address, taking 6 or 3 addresses in the two examples. With registers, if quantities named are used frequently so that loading overheads can be ignored, the number of addresses can be cut by half in the first example and eliminated in the second. The potential is that program size can be more than halved by the use of such special short addresses - this is still important consideration today given the effect of cache capacity on performance.

Although the importance of registers as being short addresses is not often stressed, there are many examples of architecture that make the point. One of the first computers to have a fast general purpose register array, the DEC PDP6 (ref. to Bell and Newall), identified 16 registers with the low 16 addresses of main storage. Architecturally, there was no difference between the 16 first words of memory and others, except that they could be referenced by short addresses; however, the user knew that the first 16 words were implemented in fast technology. Interestingly, the other extreme was taken by a contemporaneous architecture, the IBM System/360, which had one design, the model 30, in which the registers were actually stored in non-addressable part of main storage - i.e. the registers were separate from memory in the architecture but the same technology in the design.

In what follows, we will be exploring whether the speed and address-size advantages of registers can be separated in architectures/designs. If memory locations that are identified by short addresses, which we can still regard as being registers, are not necessarily held in a separate register array, we can think of the registers as being *virtual*.

Previous Virtual Register Architectures

As well as the simple PDP6 approach just mentioned there are a number of successful architectures that have made a clear distinction among the functions of registers. It is possible to have viable architectures with no registers at all, virtual or otherwise. The earliest computers often did so but nowadays the benefits of registers are widely recognized so such an extreme position is taken very carefully.

One extreme is the IBM AS400 (reference) which certainly has no concept of registers. Addresses in programs are kept reasonably short by being local to an object, but the size of an instruction with seven addresses is still XOS. However, in this machine, there is no pretence that the programs defined according to the architecture will be implemented directly but it is understood that they will be translated to a lower level of interface before execution. The question of registers is transferred to a proprietary level that we cannot comment on but believe is quite traditional in its usage of registers.

The most long-living architecture that is (almost) register free is that of the Unisys A-series (that started life as the Burroughs B6700/7700 (reference)). This is a stack machine with the stack held in main storage. Temporary operands are obtained from the top of stack rather than from registers. The implementations of the architecture in fact used high-speed registers to hold the top of stack although there were different numbers of registers in different implementations and the registers never appeared in the functional architecture (except for an instruction called PUSH that played the role of flushing the real registers, rather analogous with the IBM System/360 Purge TLB instruction - what it does is irrelevant but in certain circumstances you had better execute it if you want the architecture to work correctly!). A second use of registers in the Unisys A-series is for addressing bases - the *display registers* which are associated with the Algol implementation model which have as main advantage a short length number that allows 14-bit based addresses and 16-bit instructions. The display is explained as being registers, but there is no need at all to do so - the registers may not be loaded or stored directly in normal programming; they could equally well have been described as points on an addressing chain, with the existence of registers being again an implementation choice.

A more modern architecture that eliminates registers as a fast array is that of CRISP (Ditzel and Berenbaum 1982). Here the machine uses a stack but not for arithmetic. Rather the elements in the top stack frame must be addressed relatively by a short offset; thus the architecture does not have registers. However, the implementation certainly does make use of small very-fast cache to hold elements in the top of of the stack, which is thus used for both temporary operands and local addressing. Again, the mechanism is described along with the architecture but it is in reality only one implementation of those possible.

Another earlier architecture of interest is the Texas Instruments TMS 9900 which also used the concept of a variable frame located in main storage to serve as an area located by short addresses (reference). There seemed here to be no mention of a high-speed array as well.

Registers - Pros and Cons

We have seen that there are successful architectures that do not use registers explicitly but have some mechanism for addressing part of memory with short addresses. Why would one prefer an architecture to be one way or the other?

The impetus to remove registers comes partly from unease with duplication of effort - why should both the functional and design architectures both address the same problem - is there synergy or do the two approaches work against each other? With registers, the decision that a value is to be used more frequently than others is a dynamic one that is forced to be done statically by the compilers. If the decision is wrong then the overhead of loading/storing registers will dominate the saving. The small numbers of registers the make decision more critical and limit compactness of programs, but large numbers of registers would seem to be poor usage of resources. Then there is the problem of aliases. Because registers are a separate storage any main storage location that may be referred to indirectly cannot be safely placed in a register array, thus limiting effectiveness of registers (Hennesy and Patterson reference).

The advantages of registers are obvious. The reasons for continuing with registers are those against making architecture be of too high a level. There is often a loss of flexibility involved, for example, often there is only one block of memory that can be addressed with short addresses or we are stuck with some model of addressing, such as Algol with the Unisys approach, that may not be suited to our purposes. Then, in these pipelined days, there is great concern about performance. Lampson, in his 1982 paper addressed cogently the question *Why not just a cache?:*

"A register bank is faster than a cache, both because is is smaller, and because the addressing mechanism is much simpler. Designers of high-performance processors have typically found that it is possible to read one register and write another in a a single cycle, while two cycles are needed for a cache access. It is not too hard to build a cache which can accept a reference every cycle, but the latency is still two cycles. Also, since there are not too many registers it is feasible to duplicate or triplicate them, so that several registers can be read out simultaneously."

"Because they are faster, registers have more bandwidth, especially if they are duplicated. But more important, storing frequently accessed locals in registers frees up cache bandwidth for more random references. Half or more of all data memory references may be to local variables, Removing this burden from the cache effectively doubles its bandwidth."

"The locality of references to local variables is so much better than the locality of data referenced in general that another level of memory hierarchy which exploits this locality is highly worthwhile. Since the compiler can control quite well how these references are made, the hardware supporting this level can be much simpler than a general- purpose cache, which must fully support the simple read-write properties of storage in general."

"As a corollary of the last point , the simple ways in which local variables are addressed (nearly always by a constant displacement in an instruction) makes the addressing logic for a register bank both simple and fast. ..... No comparators, associative lookup or other complication is needed."

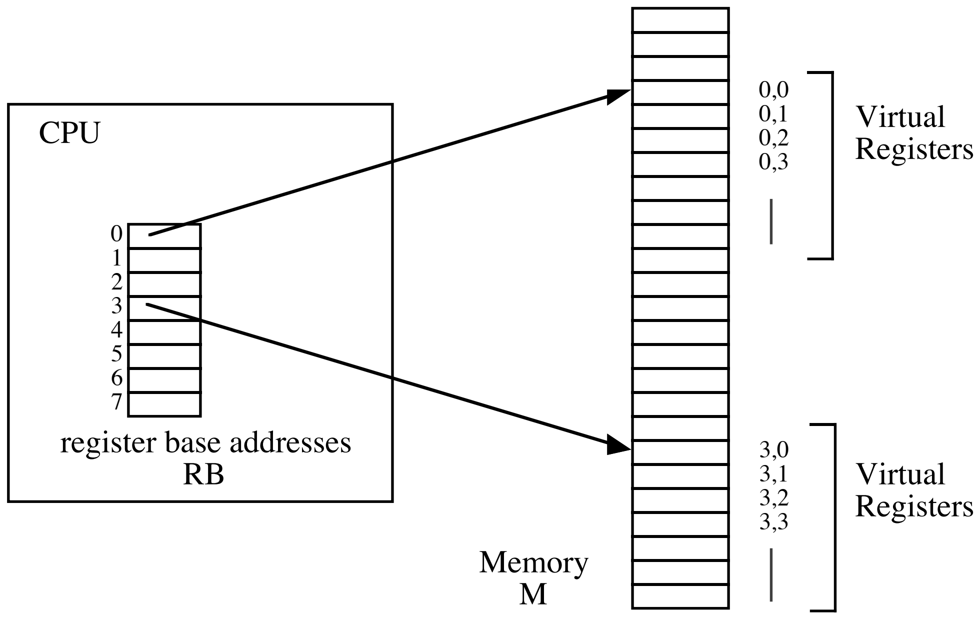
The Virtual Register Cache

The arguments of Lampson above are directed mainly towards showing that it is not reasonable to combine cache and registers. Completely separating the issues sidesteps the majority of the objections. We will provide a means of using short addresses as aliases for main storage, with a relatively large number being available to make the allocation issue less critical. The locations designated by short addresses will be our virtual registers. The architecture will retain flexibility by allowing the virtual registers to be used for general purposes and providing multiple banks of them.

The implementation, rather than the architecture, will provide a small, separate, very-fast, virtual-register cache for the short addressed items. The consistency of aliased storage will be maintained automatically. By outlining an example architecture, we intend to show is that this approach is feasible. We are Interested in arguing qualitative feasibility, rather than finding best approach, hence we will not undertake a quantitative analysis of performance as there are many parameters to be specified. However, we will give qualitative arguments that the performance can be made reasonable.

Example Functional Architecture

The computer has a set of 8 (e.g.) "register base addresses". These can themselves be considered to be registers in that they are subject to instructions for loading and storing . Each register base address identifies in main storage a block of 32 (e.g.) virtual registers that may thus be specified with an 8-bit short address comprising two fields - register base number (3 bits), register number (5 bits)). Thus there may be 256 virtual registers identified at any time,however, as blocks may be activated and deactivated at will by changing a register base address, many more memory locations may be regarded as inactive virtual registers.



Instructions refer to main storage by adding the displacement field of 8 bits (e.g) from the instruction to a virtual register used as a base to generate the full 32-bit (e.g.) address of a 32-bit (e.g.) word. The example architecture chosen is symmetric (370-style) rather than load/store (reference to paper on relative performance of each). Load/store architectures involve some difficulties that will be discussed later.

The format of a typical instruction is:

	vr1		vr2		
opcode	b1	r1	b2	r2	disp2
8	8		8		8

If the opcode is "add" then this would be interpreted as :

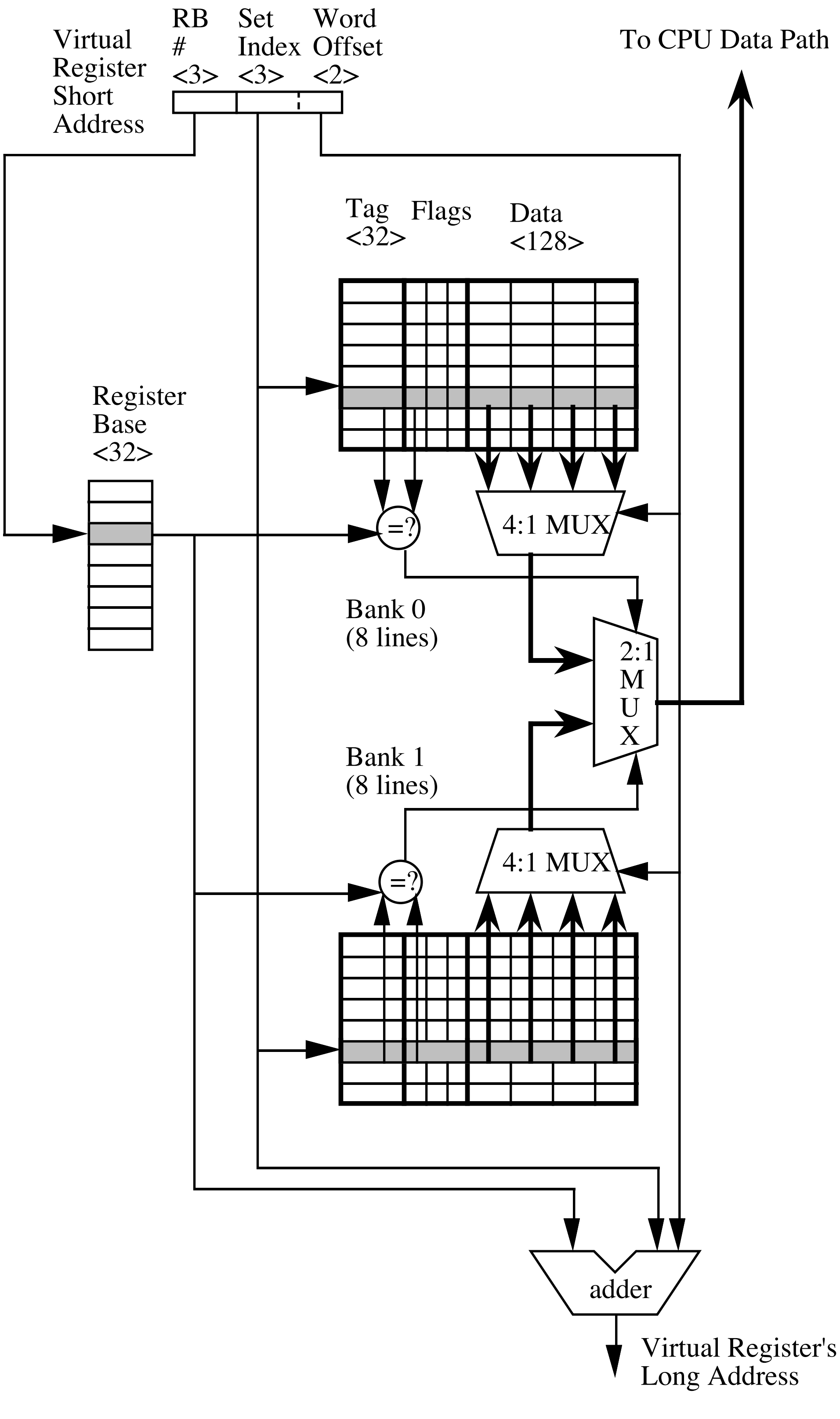
$$M[RB[b1]+r1] := M[M[RB[b2]+r2]+disp2] + M[RB[b1]+r1]$$

Apart from details of what each instruction does, the functional architecture definition is complete at this stage. Note that there is no mention of virtual registers as being held in a separate memory. We want to show that active registers can be migrated automatically to a register cache without compromising the integrity of the storage - the usage of a full address to refer to a register or any short address shall give identical results as would be expected from the above definition. However, in order to do so we will have to compromise on performance - we will accept that there is no requirement that multiple aliases for the one location be all used frequently.

The implementation concept, non-pipelined.

The idea here is that there are two levels of cache before main storage. The *long-cache* is the conventional cache as for any other architecture - it is addressed associatively by the full 32-bit main storage address. The *short-cache* holds items that are addressed by short addresses and is much smaller, being implemented from register technology. The hierarchy has the inclusion property, that is, if data is present in the short cache then it is also guaranteed to be held in the long cache.

The example short cache is one that holds 8 (e.g.) sets of 2 (e.g) lines of 4 (e.g.) adjacent words each. Each line has a tag that gives the main storage address of the block containing the 4 words in the line (4-word aligned), not the address of the line itself. The register base addresses are held in 8 fast registers. The short cache is accessed using a virtual register number which has the two fields, the register base number (3 bits) and the register number (5 bits); the latter is split into two subfields, the set index (first 3 bits) and the word index (last 2 bits, identifies the word within its line). The access sequence is to use the register base number to access the register base address in parallel with a read of the line in both sets given by the set index to locate 2 tags and 2 possible registers. The tags are matched with the register base address to determine which of the register values is the one required. The register selected may be used as an operand. Alternatively it may be used as a base for referring to storage, or, if the register addressed is not present in the cache the register base address may be modified by the register number to address main storage directly.



Assuming that all virtual registers are present in the short cache, the sequence of execution of a typical instruction like the add above is as follows. The short cache is accessed to give the base address to which the displacement is added to give the long address of the second operand. This is obtained from the long cache. In parallel with the long cache access, the short cache is accessed to obtain the first operand - a note is made of its location in the register array. After arithmetic is completed, the result is written to the first operand location directly without associative match.

This procedure is interrupted whenever there is a miss in the long or short cache. Misses in the long cache are handled conventionally. Misses in the short cache require that the long address of the register be generated and the long cache accessed to obtain the required register. This is placed in the short cache which may require the least recently used short cache line to be replaced (its address is given by the short cache tag, it would be nice to know the corresponding long cache set so that it may be written directly without associative match - note that the property of inclusion guarantees that rewriting a short cache line to the long cache can never cause a main storage access, the complexity stops there!).

Long cache and consistency

Each line of the long cache could contribute to multiple short cache lines. As long as there is no overlap there is no problem in consistency or performance. When a short cache line is moved-in, the fact that there is duplication is marked in that portion of the long cache line along with a note of the actual register array location where the duplicate it is stored. Now, an attempt to access a register by its long alias can be detected when the long cache is accessed. The suggested action is to read the information from the short cache to update the long cache if the short cache is altered, then trash the short cache line. Thus there will never be two copies of data present if both are being accessed concurrently and furthermore there will never be two copies of the same data in the short cache if register blocks overlap.

Changing a register base address does not affect data stored in the short cache at all. If a register base address is moved from one register to another, the registers can still be accessed from the short cache as long as the displacement is the same. What will cause degradation is accessing the same register from two different displacements because only one may be stored actively in the short cache and activation of a new address will require that data to be moved via the long cache.

Comments on performance and feasibility

Final arguments should wait until we have seen how the virtual register cache fits-in with a pipelined implementation. However, we will find that the pipeline doesn't have much effect except on cost and complexity, so most of our performance conclusions can be drawn now in terms of the serial implementation.

Note immediately that if the addressed registers are present in the short cache then the path to access a register has been lengthened only by the time for comparison followed by the 2:1 multiplexing. This replaces an 8:1 multiplexing so costs at most an extra couple of levels of logic per instruction execution. This degradation is not so great as to rule the concept out of order.

The other first-level performance factor is that once the short-cache has been loaded, programs execute just as if a register array is being accessed. This means that, apart from the slight increase in cycle time, performance will be comparable to a conventional register computer.

There are many second-order performance and cost factors. A obvious defect is cost in terms of storage of tags. In the example, this has been kept to a moderate 25% by the 4-word lines. This overhead would make the use of shorter short-cache lines unattractive. The register base addresses also require register storage.

Another question is whether the time taken to load and restore registers is better or worse than the time taken to load and save the short cache. Short cache lines are loaded on actual demand, which should be an advantage, and they are saved to long cache when they are accessed by an alias or thrown out of short-cache for not being active and have been altered. The trade-off here is the value of a register block move versus many smaller ones needed by the short cache mechanism - this comes down to a careful selection of the short cache line size. Overall, it seems that the short cache mechanism should involve less overhead than the explicit loading and saving of registers.

The implementation outlined will cause degradation if there is an attempt made to address a virtual register frequently with a long address and a short address (which can be avoided) or with different displacements. The latter could be a problem for accessing procedure parameters, however, If the register base address array is managed as a push through stack, this is no problem.

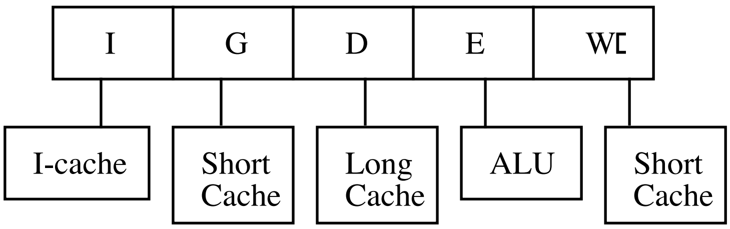
Another obvious problem with the sample design is that identical offsets beat on each other for access to the same set in the short cache so that low displacements could thrash. A design solution is to hash the set accessed (e.g. add base-register number) but this adds a level to the access. A more RISC-like approach is to ask the compiler to randomize use of displacements (where this could be a problem - it couldn't be used for procedure parameters).

The example architecture has a very small displacement field. This could be increased by saving bits elsewhere, such as reduction in register bases or register numbers. This is a trade-off that we have not explored. Note, however, that a larger displacement can be effected by having adjacent virtual registers being used as bases point to contiguous 256-word blocks of storage.

An idea that we have not explored in detail could deal with both the problems of speed of register cache access and the cost of tag storage. Rather than storing the 32-bit tag, one could store 8-bits giving the result of matching the tag against each of the register base addresses. The appropriate bit could be selected fast enough to have very little effect on the register read (only affecting the last level of selection). What would be difficult is keeping the tag-match bits current when the register base addresses are changed. A simple approach would be to take an extra cycle for all accesses when the register base address changes but how to allow values to be moved among the register base address registers?

Virtual Register Cache in a Pipeline

To be truly feasible, the virtual register cache must fit in with a pipelined implementation without causing too much difficulty. We have roughed-out an implementation involving 5 pipeline stages as follows, with little regard to the actual length of each stage:



I   Fetch instruction
G   Access short cache and generate long address
D   Access long cache
E   Execute operation
W   Write result to short cache

Operating at Speed

Consider firstly, first-order effects, when the pipeline is operating at speed with no short-cache misses. The extra access time that it takes to access the short cache compared with a register array will be amortised over the length of the pipeline so will not greatly affect cycle time. It is possible to consider speeding access by commencing the effective address addition process in duplicate before making the final selection.

As with the register array of a conventional machine, a pipelined virtual register cache will need to handle 3 accesses per cycle, two reads and one write. The write is made directly to the word addressed so it is no different to the conventional computer. However, the reads both involve an associative match.

There are two approaches that can be taken here. One is to go ahead and duplicate the read of the register base, the tags and data. This is clearly more expensive than the conventional approach. Another is to half cycle the clock and access the cache twice per cycle. The first access, to the base register is the most exigent; the second, to the operand, could take place later.

The conclusion is that in terms of operating at speed the virtual register cache should be close to matching the conventional machine, but at somewhat higher cost.

At this point the problem with a load/store architecture is apparent. This would not only require that there be three accesses per cycle, the write would need to be preceded by an associative match. The general load/store architecture thus poses considerable difficulties (OK if the register instructions involve only 2 operands).

Interlocks

There are clearly many more design dependencies with the virtual register cache than with simple registers. In essence, we are trying to move some conventional functions, loading and storing registers, into the hardware, so that more complexity is expected. What is important is to convince ourselves that the complexity is manageable and does not have threatening performance implications.

Dependencies between virtual registers may be detected based on matching the virtual register addresses, analogous with register number matching in conventional pipelines. The instructions to alter the register base addresses disrupt this process so must cause additional delays.

Most of the design dependencies are straightforward, even if complicated. For example, a miss for the virtual register base in the short cache would require an immediate interlock is as follows.

cycle	1	2	3	4	5	6	7	8	9	10	11	12
in0:	G	D	E	W								
in1:	I	G1	D1									
				G2	G3	D2	E	W				
in2:		I	I	I	G4	D	E	W				
in3:					I	G	D	E	W			

G1 - miss in the short cache for virtual base register
D1 - access long cache to obtain register value
G2 - interlock short cache for reading
G3 - read operand register
D2 - access long cache for memory operand and store short cache
G4 - short cache read with bypass if D2 write is to same register

This cause a 2-cycle delay minimum. The remainder of the short cache line has to be written to the short cache sometime - how many cycles this takes depends on how wide data paths are. If paths were 64 - bits then there is no additional delay because there is no short cache write in the cycle after D2.

There are some very complex situations that can arise. The worst is when both the operand and base virtual registers are missing from the short cache, they both require replacement, and both also cause long-cache misses, and the memory operand also causes a long cache miss. This can, however, be handled as a sequence of simpler situations.

There is one case that is significantly different from the conventional architecture. If a store is made to a long address that proves to be an alias for a short address, any subsequent instructions that refer to the short address could be invalid. There is no way of detecting this problem in time to stop subsequent instructions from accessing the short cache line. The only safe approach is to invalidate all instructions in the pipeline following the store to the long cache and cause them to be re-executed. This doesn't have much performance effect as in should not be a frequent occurrence.

Conclusions

We have outlined an approach to registers that divorces their use as short addresses from their use as fast storage. The migration of registers to fast storage is done automatically in such a way that aliases are managed correctly.

It appears that the raw first order performance of such an architecture can be made close to that of a conventional machine. The architecture probably wins over conventional machines when it comes to second-order performance effects, but the proof of this claim would require a lot more work.

The architecture described is basic and rather RISC-like. It does not subscribe to any particular model of subroutine linkage and procedure passing, but provides a general mechanism that would be made more specific in any particular implementation.

We do not argue here that the virtual register cache approach is a good direction to take, merely that it is feasible and is a concept that is worth investigating further.

References

D. R. Ditzel, & H. R. McLellan (1982):*The C machine Stack Cache: Register Allocation for Free,* Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems (March, 1982), pp. 48-56.

B. W. Lampson (1982): *Fast Procedure Calls,* pp. 66-76.

Texas Instrument Inc. (1975): #45. TS9900.